PRÁCTICA 4:

ALQUILER DE COCHES

- Juan Manuel Valenzuela González – amcgil@uma.es
- Artur Vargas Carrión – arturvargas797@uma.es
- Jorge Repullo Serrano – jorgers4@uma.es
- Eduardo González Bautista – edugb@uma.es
- Rubén Oliva Zamora – rubenoliva@uma.es
- David Muñoz del Valle – davidmunvalle@uma.es
- Alejandro Jiménez González – alejg411@uma.es

Modelado y Diseño del Software

GR1-3 | Ingeniería del Software A | Universidad de Málaga

ÍNDICE DE CONTENIDO

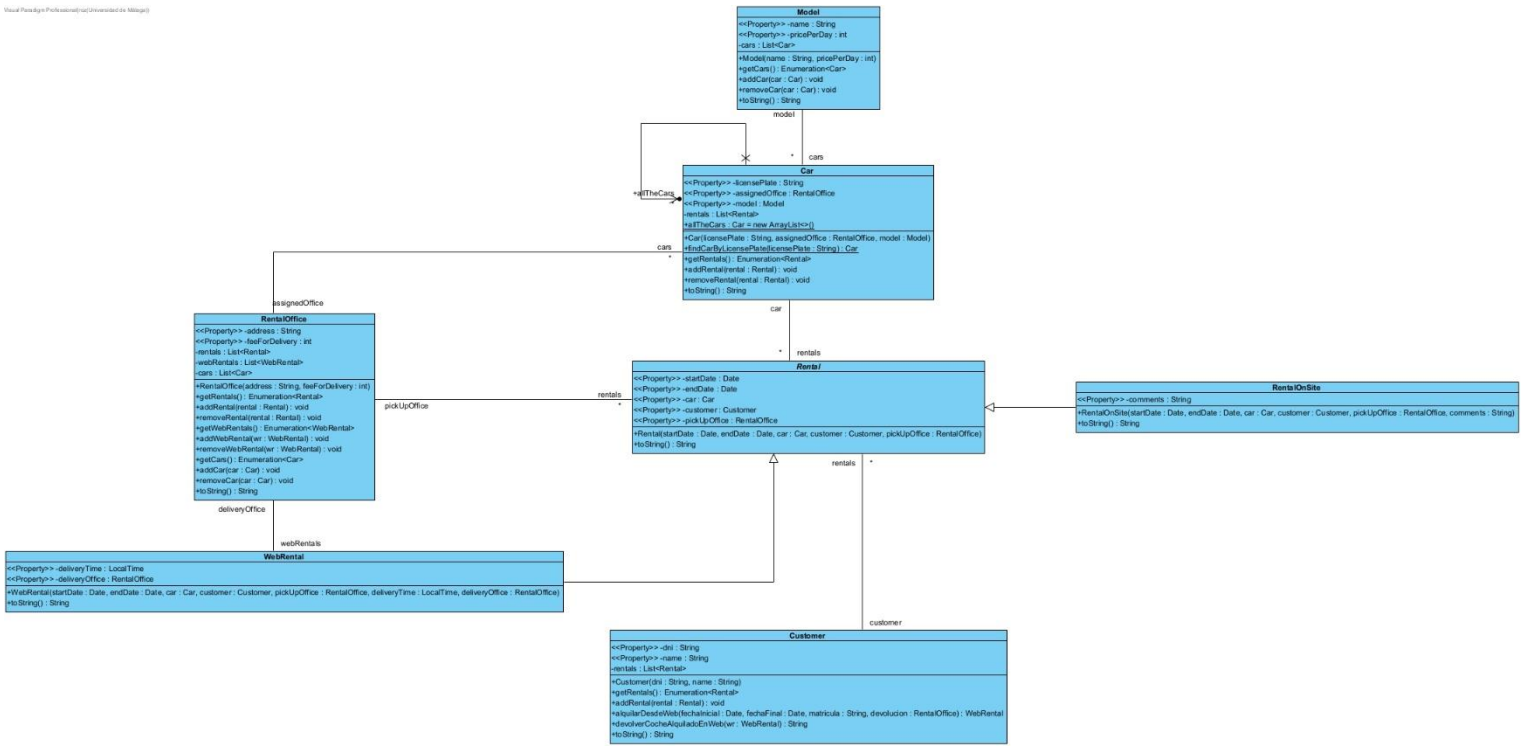
1. Base del ejercicio

En primera instancia, mostraremos la **base** que hemos desarrollado para poder realizar los ejercicios. Esta base cumple con los requerimientos mostrados en el enunciado de la práctica, incluyendo restricciones de integridad, las relaciones mostradas en el diagrama del enunciado y, en general, el código de andamiaje necesario para poder realizar los ejercicios propuestos.

Cabe destacar que, en los ejercicios propuestos, **la restricción de integridad 4**, que aparece en la clase `Customer`, **ha sido comentada** ya que los main fallan si no se ejecutan antes de las 13:00 horas. La restricción impone una limitación horaria, y el main prueba esa restricción, por lo que si se ejecuta antes de las 13:00 horas se elevará una excepción.

Como en la parte que describe la base del enunciado se mencionan **funcionalidades relacionadas con alquilar y devolver coches desde la web** (último párrafo, página 1), en la clase `Customer` hemos añadido los métodos `alquilarDesdeWeb` y `devolverCocheAlquiladoEnWeb`. Para crear estos métodos hemos necesitado definir una lista estática que almacene todos los coches, `allTheCars`, para poder encontrar un coche por su matrícula en el método `findCarByLicensePlate`. Esta lista será útil en el ejercicio 2 para poder hallar un coche sustituto.

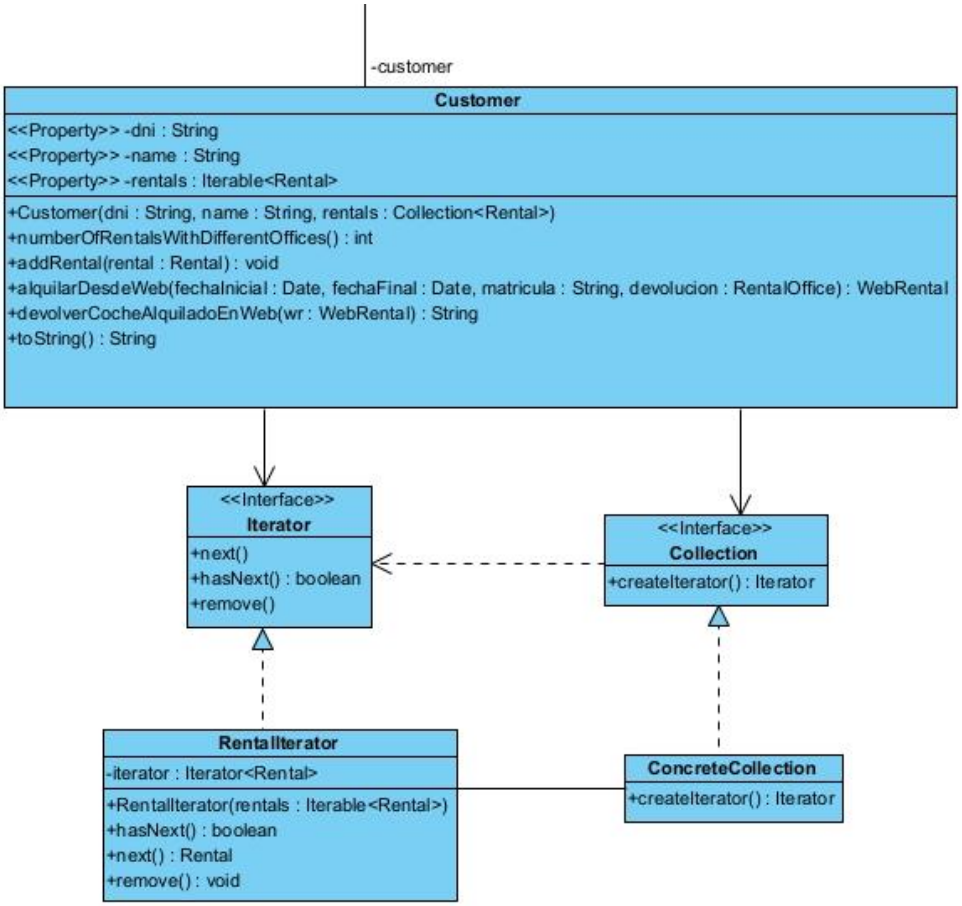
Para hacer el diagrama nos hemos ayudado de la herramienta Reverse Code que incluye Visual Paradigm, además de añadir aquello que fuese necesario y esta herramienta no proporcionase (por ejemplo, no pone las variables de tipo lista como `List<T>`). Es importante destacar que los **atributos marcados como** `<<Property>>` son aquellos que **cuentan con getters y setters dentro del código**, a pesar de que estos **no se muestren explícitamente en el diagrama UML**.



1 Base sobre la que realizar los ejercicios

2. Ejercicio 1

2.1. Elección del patrón de diseño. *Iterator*



2 Ejercicio 1. Cambios realizados en el diagrama de clases al introducir el nuevo método que usa el patrón Iterator

Se ha puesto el Visual Paradigm de forma que aparezca **similar a lo mostrado en la teoría**, aunque no hayamos hecho las interfaces Iterator y Collection, ya que hemos usado las **ya implementadas en Java**. De esta forma, nuestro ConcreteIterator será `RentalIterator`. ConcreteCollection lo hemos dejado así ya que el enunciado especifica que no sabemos qué colección se va a usar.

En Java, Collection extiende la interfaz Iterator, por lo que podemos asegurar que podremos recorrer cualquier colección que implementemos con un iterador.

Para diseñar la operación `numberOfRentalsWithDifferentOffices` en la clase `Customer`, donde no sabemos aún la estructura de datos que almacenará los alquileres, utilizaremos el **Patrón Iterador (Iterator)**. Este patrón es adecuado para este caso por las siguientes razones:

- **Abstracción del almacenamiento**: el patrón Iterador permite recorrer una colección de elementos sin exponer los detalles internos de su implementación. Como aún no sabemos qué estructura de datos se usará para almacenar los alquileres (puede ser una lista, un conjunto, una base de datos, etc.), el iterador desacopla la lógica de la operación de la representación concreta de los datos.

- **Flexibilidad en el diseño**: el cliente (la clase `Customer`) solo interactúa con el iterador para recorrer los alquileres, lo que facilita cambiar la implementación interna de la colección en el futuro sin afectar la lógica de la operación `numberOfRentalsWithDifferentOffices`.

- **Reutilización de lógica de iteración**: al encapsular el acceso a los alquileres mediante un iterador, puedes reutilizar la misma lógica de iteración en otras operaciones similares que deban procesar los alquileres de manera específica.

- **Cumplimiento del principio de responsabilidad única**: el cliente no tiene que preocuparse por cómo se almacenan los alquileres; solo utiliza la funcionalidad proporcionada por el iterador para acceder a ellos.

- **Flexibilidad en el recorrido de la colección:** al implementar la interfaz `Iterator` podemos hacer que la colección se recorra de la forma que queramos. Solo debemos cambiar el método `hasNext()` y `next()` en el `ConcreteIterator`, que en este caso es `RentalIterator`.

2.2. Código Java. *numberOfRentalsWithDifferentOffices():Integer* en la clase Customer

```java
/**
 * Método que devuelve el número de alquileres web donde las oficinas de
recogida y entrega son diferentes.
 *
 * @return Número de alquileres con oficinas diferentes.
 */
public int numberOfRentalsWithDifferentOffices() {
    RentalIterator iterator = new RentalIterator(rentals);
    int count = 0;
    while (iterator.hasNext()) {
        Rental rental = iterator.next();
        if (rental instanceof WebRental) {
            WebRental webRental = (WebRental) rental;
            if
(!webRental.getPickUpOffice().equals(webRental.getDeliveryOffice())) {
                count++;
            }
        }
    }
    return count;
}
```

El código arriba adjuntado hace referencia al método `numberOfRentalsWithDifferentOffices()` implementado en la clase `Customer`. Este método hace uso de un ConcreteIterator que definimos como una clase aparte `RentalIterator`, que implementa la interfaz `Iterator<Rental>` para definir sus métodos `hasNext()`, `next()` y `remove()`. El remove se ha definido como operación no soportada, ya que entendemos que se debe guardar el histórico de alquileres, y que este historial no debe poder ser borrado.

En el método a implementar lo único que se hace es:

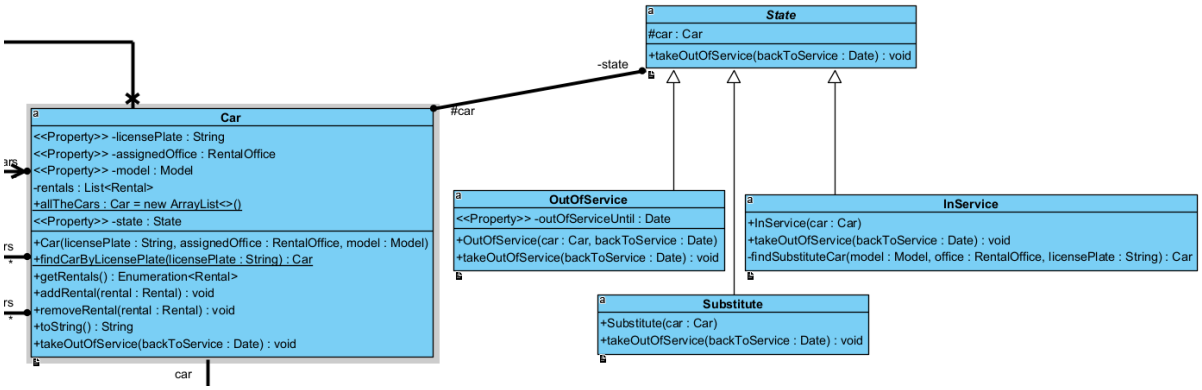
1. Iterar sobre la estructura de datos que almacena los alquileres, que puede ser una cualquiera.
2. **Comprobar que el alquiler que estamos mirando es un WebRental** y no un RentalOnSite.
3. Comprobar que la **oficina de recogida** de este WebRental es **distinta** a la oficina designada como **oficina de entrega**. Si lo es, el contador lo registrará.
4. Tras terminar de iterar sobre la estructura de datos que almacena los alquileres, la función devolverá el valor de este contador.

También adjuntamos el código de la clase `RentalIterator`, que implementa la interfaz Iterator de Java y redefine sus métodos según con lo solicitado en el enunciado del ejercicio. Como se puede observar, no dista mucho de la implementación predeterminada de Iterator, pero la existencia de esta clase permite modificar el orden de iteración y otras cuestiones de esta índole solo modificando la redefinición de los métodos `hasNext()` y `next()`.

```
import java.util.Iterator;

/**
 * Clase que implementa un iterador concreto para recorrer la colección de
alquileres.
 * Permite iterar sobre una colección de objetos de tipo Rental sin exponer
 * los detalles internos de la colección.
 */
class RentalIterator implements Iterator<Rental> {
    private Iterator<Rental> iterator;

    /**
     * Constructor de la clase RentalIterator.
     *
     * @param rentals Colección iterable de objetos Rental sobre la cual se
va a iterar.
     */
    public RentalIterator(Iterable<Rental> rentals) {
        this.iterator = rentals.iterator();
    }

    /**
     * Comprueba si hay más elementos en la colección que iterar.
     *
     * @return true si hay más elementos en la colección; false en caso
contrario.
     */
    @Override
    public boolean hasNext() {
        return iterator.hasNext();
    }

    /**
     * Devuelve el siguiente elemento de la colección.
     *
     * @return El siguiente objeto Rental de la colección.
     */
    @Override
    public Rental next() {
        return iterator.next();
    }

    /**
     * Operación no soportada para este iterador.
     * Se lanza una excepción si se intenta usar.
     *
     * @throws UnsupportedOperationException Siempre que se llame a este
método.
     */
    @Override
    public void remove() {
        throw new UnsupportedOperationException("Remove operation is not
supported.");
    }
}
```

3. Ejercicio 2

3.1. Elección del patrón de diseño. *State*



3 Ejercicio 2. Cambios realizados en el diagrama de clases al introducir el nuevo método que usa el patrón State

Para diseñar el paso de servicio a fuera de servicio de los coches hemos considerado ambos estados (y la posibilidad de ser un coche sustituto) como implementaciones de una clase abstracta `State` usando el **Patrón State.** Las ventajas de usar este patrón para este ejercicio específicamente son:

- **Encapsulamiento del comportamiento de los estados:** el funcionamiento de cada estado se encapsula en diferentes clases, lo que acaba con la necesidad de crear una estructura compleja dentro de la clase `Car`, derivada de largas y repetitivas sentencias if en cada uno de sus métodos.

- **Principio de responsabilidad única:** cada una de las clases que representa un estado contiene una única responsabilidad (la de definir el comportamiento del coche en ese estado específico).

- **Flexibilidad para agregar estados nuevos:** crear nuevos estados es muy sencillo, solamente se tiene que crear una nueva clase que implemente la interfaz `State` (o clase abstracta si no se usa `Context`). Esto deriva de las dos ventajas anteriores ya explicadas.

- **Facilidad en la transición de estados:** la transición entre los estados es sencilla debido al método `takeOutOfService(backToService : Date)` de la interfaz `State` delegando dicha responsabilidad a la clase de estado actual.

Cabe destacar que **no se ha introducido una clase similar a la clase `Context` de la teoría por simplicidad**. En los ejemplos de la aplicación de este patrón en la teoría no se usa una clase `Context`, aunque tendría sentido si quisiésemos extender los estados a otro tipo de clases (como una hipotética nueva clase `Moto`) solo teniendo que añadir un atributo de tipo `Context` a esa clase, sin tener que definir de nuevo los métodos. En el siguiente ejercicio se usa un patrón que también implementa una clase `Context` (no exactamente de la misma manera, pero la idea es la misma), que también tiene una interfaz e implementaciones concretas. Mostramos en ese apartado la forma en la que se procedería con la clase `Context` introducida.

3.2. Código Java. *takeOutOfService(backToService : Date) : void*

La responsabilidad de cambio de estado es delegada al estado actual del coche mediante el uso del método `takeOutOfService(backToService : Date)`

```
/**
 *  Cambia  el  estado  del  coche  a  "fuera  de  servicio".
 *
 * @param backToService Fecha hasta la cual estará fuera de servicio.
 */
public      void      takeOutOfService(Date      backToService)      {
    state.takeOutOfService(backToService);
}/**
 * Clase abstracta que define el estado de un coche en el sistema.
 * Proporciona una operación común que todos los estados deben implementar.
 */
public           abstract           class           State           {
    protected                          Car                          car;
    /**
     *  Cambia  el  estado  de  un  coche  a  "Fuera  de  Servicio".
     *
     * @param backToService Fecha hasta la cual el coche estará fuera de
servicio.
     *                      Cada implementación del estado manejará esta
operación
     *                                          de  manera  específica.
     */
    public       void       takeOutOfService(Date       backToService){
        throw new IllegalStateException("El método no ha sido definido");
    }
}
```

Cada estado implementa el método `takeOutOfService(backToService : Date)` de la clase abstracta `State`, de forma que tengan un funcionamiento correcto en base al estado en el que se encuentran.

Para la clase `InService` tenemos:

```
/**
 * Clase que representa el estado "En Servicio" de un coche.
 */
public class InService extends State {

    /**
     * Constructor de la clase InService.
     *
     * @param car El coche que se encuentra en el estado "En Servicio".
     */
    public InService(Car car) {
        this.car = car;
    }

    /**
     * Cambia el estado del coche a "Fuera de Servicio".
     * Si se encuentra un coche sustituto disponible, se asignará como
sustituto.
     *
     * @param backToService Fecha hasta la cual el coche estará fuera de
```

```
servicio.
     */
    @Override
    public void takeOutOfService(Date backToService) {
        // Buscar un coche sustituto
        Car substituteCar = findSubstituteCar(car.getModel(),
car.getAssignedOffice(), car.getLicensePlate());
        if (substituteCar != null) {
            substituteCar.setState(new Substitute(car));
            System.out.println("Coche sustituto asignado: " +
substituteCar.getLicensePlate());
        } else {
            System.out.println("No hay coches sustitutos disponibles.");
        }

        // Cambiar el estado del coche a "Fuera de Servicio"
        car.setState(new OutOfService(car, backToService));
        System.out.println("Coche marcado como fuera de servicio hasta: " +
backToService);
    }

    /**
     * Busca un coche sustituto disponible que pertenezca al mismo modelo y
oficina.
     *
     * @param model Modelo del coche original.
     * @param office Oficina asignada al coche original.
     * @return Un coche sustituto disponible, o null si no se encuentra
ninguno.
     */
    private Car findSubstituteCar(Model model, RentalOffice office, String
licensePlate) {
        return car.allTheCars.stream()
                .filter(c -> car.getModel().equals(model) &&
                        c.getAssignedOffice().equals(office) &&
                        c.getState() instanceof InService &&
                        !c.getLicensePlate().equals(licensePlate))
                .findFirst()
                .orElse(null);
    }
}
```

Es relevante destacar que para hallar el coche sustituto se ha usado la variable estática `allTheCars`, donde se almacenan todos los coches, que fue creada en la base para añadir los métodos descritos en el enunciado.

Para `OutOfService` tenemos:

```java
/**
 * Clase que representa el estado "Fuera de Servicio" de un coche.
 */
public class OutOfService extends State {
    private Date outOfServiceUntil;

    /**
     * Constructor de la clase OutOfService.
     *
     * @param car El coche que se encuentra en estado "Fuera de Servicio".
     * @param backToService Fecha hasta la cual el coche estará fuera de
servicio.
     */
    public OutOfService(Car car, Date backToService) {
        this.outOfServiceUntil = backToService;
        this.car = car;
    }

    /**
     * Indica que el coche ya se encuentra en estado "Fuera de Servicio".
     *
     * @param backToService Fecha propuesta para el estado "Fuera de
Servicio". No tiene efecto,
     *                      ya que el coche ya está fuera de servicio.
     */
    @Override
    public void takeOutOfService(Date backToService) {
        System.out.println("Este coche ya está fuera de servicio");
    }

    /**
     * Obtiene la fecha hasta la cual el coche estará fuera de servicio.
     *
     * @return La fecha de finalización del estado "Fuera de Servicio".
     */
    public Date getOutOfServiceUntil() {
        return outOfServiceUntil;
    }

    /**
     * Actualiza la fecha hasta la cual el coche estará fuera de servicio.
     *
     * @param outOfServiceUntil Nueva fecha de finalización del estado
"Fuera de Servicio".
     */
    public void setOutOfServiceUntil(Date outOfServiceUntil) {
        this.outOfServiceUntil = outOfServiceUntil;
    }
}
```

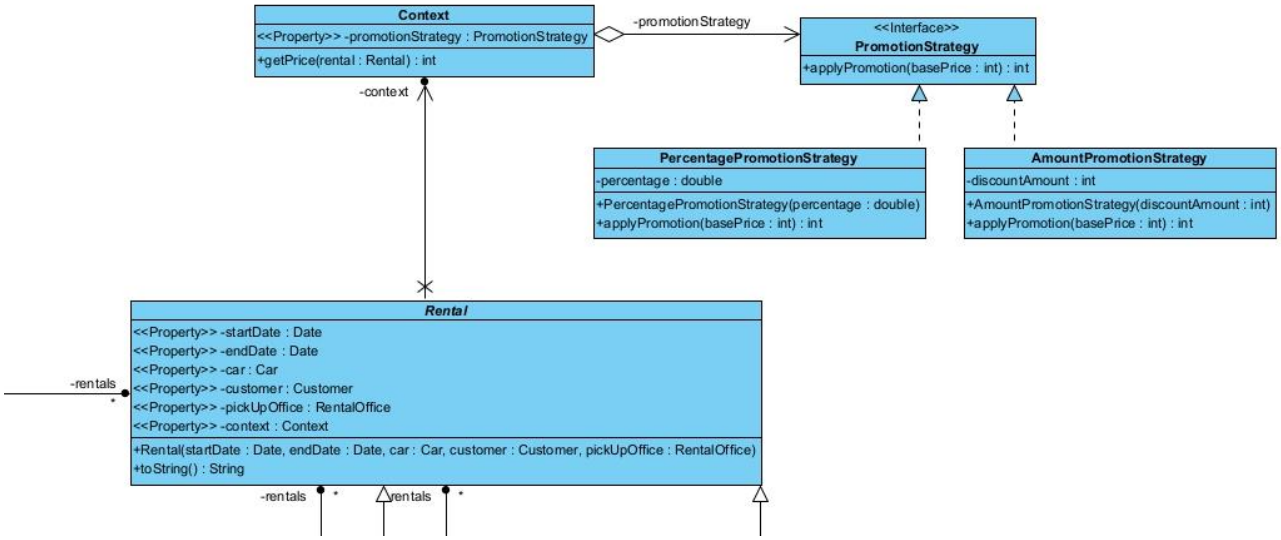
Y, finalmente, para `Substitute` tenemos:

```
/**
 * Clase que representa el estado "Sustituto" de un coche.
 * Un coche en estado "Sustituto" no puede ser marcado como "Fuera de
Servicio".
 */
public class Substitute extends State {

    /**
     * Constructor de la clase Substitute.
     *
     * @param car El coche que se encuentra en estado "Sustituto".
     */
    public Substitute(Car car) {
        this.car = car;
    }

    /**
     * Indica que un coche en estado "Sustituto" no puede ser puesto fuera
de servicio.
     *
     * @param backToService Fecha propuesta para el estado "Fuera de
Servicio". No tiene efecto,
     *                      ya que los coches sustitutos no pueden cambiar
a este estado.
     */
    @Override
    public void takeOutOfService(Date backToService) {
        System.out.println("Este coche es sustituto, no puede ponerse fuera
de servicio");
    }
}
```

Cada una de estas clases encapsula la lógica del método a implementar, de tal forma que se aplique correctamente el patrón State.

4. Ejercicio 3

4.1. Elección del patrón de diseño. *Strategy*



4 Ejercicio 3. Cambios realizados en el diagrama de clases al introducir el nuevo método que usa el patrón Strategy

Para calcular el precio del alquiler con promociones en la operación `getPrice` de la clase `Rental`, utilizaremos el Patrón `Strategy`. Este patrón es adecuado para este caso por las siguientes razones:

- **Encapsulación de las promociones:** el patrón `Strategy` permite encapsular el cálculo de las promociones en clases específicas. Esto es ideal para manejar diferentes tipos de promociones (como por cantidad fija o porcentaje) de manera independiente y sin mezclar la lógica de cálculo con el resto del código.

- **Cumplimiento del Principio Abierto/Cerrado (OCP):** este patrón permite añadir nuevas promociones sin modificar las clases existentes. Si en el futuro se define un nuevo tipo de promoción, bastará con crear una nueva estrategia concreta que implemente la interfaz común.

- **Flexibilidad en el diseño:** el contexto de `Rental` delega el cálculo de la promoción a una estrategia, lo que permite cambiar o aplicar dinámicamente diferentes promociones sin alterar la lógica del cálculo del precio base. Además, esto facilita el mantenimiento y la reutilización en otras partes del sistema.

- **Separación de responsabilidades:** cada promoción es responsable únicamente de su cálculo. Esto cumple con el principio de responsabilidad única, ya que el contexto de `Rental` solo se encarga de calcular el precio base y delega las promociones al patrón `Strategy`.

- **Utilización de la clase Context:** para mejorar la escalabilidad del programa usamos la clase `Context` (esta clase define el método `getPrice()` y regula su funcionamiento dependiendo la `promotionStrategy` que tenga definida), que permite introducir clases similares sin redefinir de nuevo los métodos de cambio de estado, solo introduciendo una variable de tipo `Context` en la clase que debe

implementar las estrategias. Por ejemplo, si además de una clase `Rental`, tuviéramos una clase `Purchase`, para implementar las promociones que ya existen solo deberíamos añadir un atributo de tipo `Context` a la clase `Purchase` (además de redefinir la lógica asociada a las propiedades inherentes al concepto de "comprar", en contraposición al concepto de "alquilar").

4.2. Código Java. *getPrice():Integer*

Este método pertenece a la clase `Context`. Las clases relacionadas con el alquiler de vehículos contienen el método `getPrice()` definido en dicha clase.

```
import java.util.Date;

public class Context {
    private PromotionStrategy promotionStrategy;

    public PromotionStrategy getPromotionStrategy() {
        return promotionStrategy;
    }

    public void setPromotionStrategy(PromotionStrategy promotionStrategy) {
        this.promotionStrategy = promotionStrategy;
    }

    /**
     * Obtiene el precio final del alquiler
     *
     * @return Precio final
     */
    public int getPrice(Rental rental) {

        long diffInMillis = rental.getEndDate().getTime() -
rental.getStartDate().getTime();
        int rentalDays = (int) (diffInMillis / (1000 * 60 * 60 * 24));
        int precioFinah = rentalDays *
rental.getCar().getModel().getPricePerDay();
        if (promotionStrategy != null) {
            precioFinah = promotionStrategy.applyPromotion(precioFinah);

        }
        if(rental instanceof  WebRental){
            precioFinah +=
((WebRental)rental).getDeliveryOffice().getFeeForDelivery();
        }

        return precioFinah;
    }
}
```

Clase `PercentagePromotionStrategy`

```
public class PercentagePromotionStrategy implements PromotionStrategy {
    private final double percentage;

    public PercentagePromotionStrategy(double percentage) {
        this.percentage = percentage;
    }

    @Override
    public int applyPromotion(int basePrice) {
```

```
        return (int) (basePrice * (1 - percentage / 100));
    }
}
```

Clase `AmountPromotionStrategy`

```
public class AmountPromotionStrategy implements PromotionStrategy {
    private final int discountAmount;

    public AmountPromotionStrategy(int discountAmount) {
        this.discountAmount = discountAmount;
    }

    @Override
    public int applyPromotion(int basePrice) {
        return Math.max(basePrice - discountAmount, 0); // Ensure no
negative price
    }
}
```